AMENDMENTS TO THE SPECIFICATION

Please replace the paragraph on page 1, lines 24-27 and page 2, lines 1-4 with the following amended paragraph:

If an application crashes or is not working as expected, an inquiry may be made as to why that application crashes or is not working as expected. Examples of information that may be useful in such an inquiry include return values and exceptions. A return value is data provided from a given method upon exit from that method. For example, in the ~~Java~~ JAVA programming language the "return" statement is used to exit from the current method and proceed with the statement that follows the original method call. There are two forms of return: one returns a value and one doesn't. To return a value, the value or an expression that calculates the value is placed after the "return" keyword. (JAVA is a trademark of SUN Microsystems, Incorporated.)

Please replace the paragraph on page 2, lines 5-14 with the following amended paragraph:

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Many kinds of errors can cause exceptions. They range from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a ~~Java~~ JAVA method, the method creates an exception object and hands it off to the run time system. The exception object contains the information about the error, including its type and, possibly, the state of the program when the error occurred. The run time system is then responsible for finding some code to handle the error. In ~~Java~~ JAVA terminology, creating an exception object and handing it to the run time system is called throwing an exception.

Please replace the paragraph on page 3, lines 1-20 with the following amended paragraph:

The first step in constructing an exception handler is to enclose the statements that might throw an exception within a "Try" block. Exception handlers are associated with a "Try" block by providing one or more "Catch" blocks directly after the "Try" block. Each "Catch" block is an exception handler and handles the type of exception indicated by its argument. Exception handling may optionally include a means for cleaning up before allowing control to be passed to a different part of the program. This can be done by enclosing the clean up code within a "Finally" block. ~~Java's~~ The "Finally" block in the JAVA code provides a mechanism that allows a method to clean up after itself regardless of what happens within the "Try" block. Consider the following pseudocode as an example:

```
public foo()
     {
     try  {
          [~~Java~~ JAVA instructions]
     } catch (ExceptionType name) {
          [code for "Catch" block]
     } finally {
          [code for "Finally" block]
     }
}
```

Please replace the paragraph on page 3, lines 21-26 with the following amended paragraph:

The method foo() is comprised of logic implemented by [~~Java~~ JAVA instructions]. These [~~Java~~ JAVA instructions] are enclosed in a "Try" block. If an exception of type ExceptionType occurs while performing the [~~Java~~ JAVA instructions], then the "Catch" block is performed. Regardless of whether the method runs without error or there is an exception, the "Finally" block will be performed. The "Finally" block is performed after the "try" block and the "Catch" handler.

Please replace the paragraph on page 6, lines 7-14 with the following amended paragraph:

One implementation of the present invention operates on ~~Java~~ JAVA code. For example purposes, the remaining portions of this document provide examples using ~~Java~~ JAVA code. However, the present invention applies to other programming languages and formats as well. Furthermore, the examples herein make use of the term "method," which has a specific meaning in reference to the ~~Java~~ JAVA programming language. For purposes of this document, "method" includes a ~~Java~~ JAVA method as well as other sets of instructions such as procedures, functions, routines, subroutines, sequences, processes, etc.

Please replace the paragraph on page 7, lines 2-6 with the following amended paragraph:

Figure 1 depicts an exemplar process for modifying an application's bytecode. Figure 1 shows Application 2, Probe Builder 4, Application 6 and Agent 8. Application 6 includes probes, which will be discussed in more detail below. Application 2 is the ~~Java~~ JAVA application before the probes are added. In embodiments that use programming languages other than ~~Java~~ the JAVA programming language, Application 2 can be a different type of application.

Please replace the paragraph on page 7, lines 7-12 with the following amended paragraph:

Probe Builder 4 modifies the byte code for Application 2 to add probes and additional code to Application 2 in order to create Application 6. The probes measure specific pieces of information about the application without changing the application's business logic. Probe Builder 4 also installs Agent 8 on the same machine as Application 6. Once the probes have been installed in the bytecode, the ~~Java~~ JAVA application is referred to as a managed application.

Please replace the paragraph on page 11, lines 11-27 and page 12, lines 1-2 with the following amended paragraph:

In one embodiment, the object code that is being modified is stored in a class data structure according to the ~~Java~~ JAVA Virtual Machine Specification. Figure 3 is a symbolic representation of the class data structure which holds the code for a class. The term code is used to refer to all of the instructions, variables, definitions, pointers, addresses etc, that are stored in a class file and/or a class data structure. Magic item 202 supplies the magic number identifying the class file. The values of minor version 204 and major version 206 are the minor and major version numbers of the compiler that produced the class file. Constant pool count item 208 provides the number of entries in the constant pool. Constant pool 210 is a table of variable length structures representing various string constants, class names, field names, integers, floating point numbers and other constants that are referred to within the class file structure and its substructures. The value of access flags item 212 is a mask of modifiers used with class and interface declarations. The access flags modifiers are public, final, super, interface and abstract. The value of this class item 214 must be a valid index into the constant pool table. The constant pool entry at that index must be a structure representing the class or interface defined by this class file. For a class, the value of superclass item 216 either must be zero or must be a valid index into the constant pool. If the value of the superclass item is nonzero, the constant pool entry at that index must be a structure representing the superclass of the class defined by this class file.

Please replace the paragraph on page 12, lines 11-18 with the following amended paragraph:

The value of methods count item 226 provides the number of method_info structures in methods table 228. Each entry in methods table 228 must be a variable-length method_info structure providing a complete description of the ~~Java~~ JAVA Virtual Machine code for a method in the class or interface. The method_info structures represent all methods, both instance methods and, for classes, class (static) methods, declared by this class or interface type.

Please replace the paragraph on page 12, lines 19-22 with the following amended paragraph:

The value of the attributes count item 230 provides the number of attributes in attributes table 232. Each entry in attributes table 232 must be a variable-length attribute structure. A class data structure can have any number of attributes associated with it. More information about ClassFile formats and the ~~Java~~ JAVA Virtual Machine can be found in The ~~Java~~ JAVA Virtual Machine Specification, Tim Lindholm and Frank Yellin, Addison-Wesley, 1997, incorporated herein by reference.

Please replace the paragraph on page 13, lines 6-14 with the following amended paragraph:

The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a method. The access_flags modifiers are public, private, protected, static, final, synchronized, native, abstract and strict. The value of the name_index item must be a valid index into the constant pool. The constant pool entry at that index is a valid ~~Java~~ JAVA method name. The value of the descriptor_index item must be a valid index into the constant pool representing a valid ~~Java~~ JAVA method descriptor. The value of the attributes_count item indicates the number of additional attributes of this method. Each value of the attributes table must be a variable-length attribute structure. A method can have code attributes and exception attributes.

Please replace the paragraph on page 13, lines 15-18 with the following amended paragraph:

The Code attribute is a variable-length attribute used in the attributes table of method_info structures. A Code attribute contains the ~~Java~~ JAVA virtual machine instructions for a single ~~Java~~ JAVA method, instance initialization method or class interface initialization method.

Please replace the paragraph on page 16, lines 9-27 and page 17, lines 1-10 with the following amended paragraph:

In step 302 of Figure 5, the system accesses the beginning of the byte code for a particular method. In one implementation, step 302 includes accessing the first element in the code array of the code_attribute of the method_info structure. In step 304, the indices for the byte code are adjusted. That is, the system knows it needs to add code and knows how many bytes the code occupies. These bytes will be added to the array. Therefore, the remaining instructions need to be moved (change index) to make room for the new instructions. For example, if the start code needs 8 bytes, then the indices for the original code needs to be adjusted to reflect the displacement by 8 bytes. Additionally, all references to byte code within an instruction must also be adjusted; for example, the pointer reference for a jump or branch instruction must be adjusted. In step 306, new start code is added to the code array. An example of start code is code that starts a timer, tracer or other profiling mechanism. Some embodiments do not use start code. In step 308, the system accesses the end of the byte code, which in one embodiment is the end of the code array. In step 310, new exit byte code is added at the end of the code array. An example of exit code is code used to stop a timer, tracer or other profiling mechanism, as well as code to record information and/or perform action at the conclusion of a method. Some embodiments include adding all or a portion of the exit byte code at locations other than the end of the byte code. Step 310 also includes adding additional byte code to the code array that allows the return values and exceptions to be accessed. In step 312 of Figure 5, the exception table is adjusted. That is, because the indices of the byte codes changed, the values of start_pc, end_pc and handler_pc may need to be changed for existing exception handlers. Step 312 can also include adjusting other tables in the class file, as appropriate to the particular implementation. In step 314, a new entry is added to the exception table. This new entry correlates to the new AFinally@ block. The new entry has a catch_type of zero, indicating it is for all exceptions. Additionally, the new entry in the exceptions table will be added to the end of the exceptions table. The start_pc and end_pc of the new entry are set to include the original ~~Java~~ JAVA instructions for the method being modified. The value of the handler_pc for the new entry would point to the new byte code added in step 310.

Please replace the paragraph on page 17, lines 11-19 with the following amended paragraph:

Figure 6 is a flowchart describing more detail of the process of adding new exit byte code and the additional byte code for accessing return values and exceptions (step 310 of Figure 5). In step 402, new code is added to the code array to account for the situation where there is no exception. That is, the original ~~Java~~ JAVA method instructions are performed without any exceptions. After the original instructions are performed, the "Finally" block should be executed. The original code is modified to include a jump to the byte code subroutine for the AFinally@ block. Additionally, new byte code is added to access the return value and pass that return value to an object (or other structure, process, entity, etc.) for storage, statistical or other purposes.